Optimising Sparse Matrix Vector Multiplication for Large Scale FEM problems on FPGA

Paul Grigoraş*, Pavel Burovskiy*, Wayne Luk*, Spencer Sherwin†
* Department of Computing, Imperial College London
† Department of Aeronautics, Imperial College London
Email: paul.grigoras09@imperial.ac.uk

***Abstract*—Sparse Matrix Vector multiplication (SpMV) is an important kernel in many scientific applications. In this work we propose an architecture and an automated customisation method to detect and optimise the architecture for block diagonal sparse matrices. We evaluate the proposed approach in the context of the spectral/hp Finite Element Method, using the local matrix assembly approach. This problem leads to a large sparse system of linear equations with block diagonal matrix which is typically solved using an iterative method such as the Preconditioned Conjugate Gradient. The efficiency of the proposed architecture combined with the effectiveness of the proposed customisation method reduces BRAM resource utilisation by as much as 10 times, while achieving identical throughput with existing state of the art designs and requiring minimal development effort from the end user. In the context of the Finite Element Method, our approach enables the solution of larger problems than previously possible, enabling the applicability of FPGAs to more interesting HPC problems.**

I. Introduction

Sparse Matrix Vector multiplication (SpMV) is an important kernel in many scientific applications such as the Finite Element Method [1]. Typical challenges associated with the SpMV kernel are related to the indirect memory access pattern which leads to poor resource utilisation and compute efficiency on modern architectures [2], [3]. In this regard, FPGAs may have a considerable advantage compared to general purpose architectures: the fine degree of customisation available can be used to directly and carefully orchestrate data movement on and off-chip resulting in good performance on the SpMV kernel [4]–[6]. Furthermore, when using FPGAs there is a great potential for application and domain driven customisation: wordlengths, reduction circuits, memory controller infrastructure can all be optimised to direct resources to the most critical component [7]–[11].

However, fine-tuned FPGA implementations can take months to develop and seldom incorporate all state of the art techniques required to exceed the raw performance of CPU and GPU systems [6]. The challenge therefore is to automate the customisation process, and explore the design space effectively while capturing more problem specific information, such as sparsity pattern. A higher level, automated approach to customisation can make high-performance FPGA cores directly available to end users with limited to no effort.

In this work we propose an automated method to detect and optimise a sparse matrix vector multiplication unit for a floating point, variable size dense block diagonal matrix vector multiplication. Our approach is based on the following contributions:

1) optimised architecture and implementation on a commercial FPGA system for mixed-precision sparse block-diagonal matrix vector multiplication;
2) resource constrained performance model to guide the tuning process of the provided architecture on user provided matrix instances;
3) automated method for customising the proposed architecture based on a given matrix and the resource constrained performance model;
4) evaluation of the proposed method on a classical HPC problem: the Finite Element Method (FEM). At the core of the method an iterative linear solver is used which requires a large number of matrix-vector multiplications to be performed where the sparse matrix has dense block diagonal structure.

II. Background and Related Work

Sparse matrix vector multiplication (SpMV) refers to the multiplication of a *sparse* matrix A to a vector x to produce a result vector b: $A \times x = b$. A matrix is considered *sparse* if sufficient entries are zero and this fact can be exploited by adequate representation and algorithms to reduce the storage size or reduce the execution time of various operations [12].

The most generic optimised storage format for sparse matrices is Compressed Sparse Row (CSR) storage which encodes the value and position of each nonzero in the matrix. While efficient and generic, CSR does not take advantage of any properties of the sparsity pattern. For instance, matrices with banded structure or where nonzeros are grouped in (almost) dense blocks occur often in practice. This insight can be used to create more optimised block-based storage formats [13], where only the position of nonzero blocks needs to be stored. This reduces the amount of metadata to store and increases the computational efficiency due to the dense local structure. In this work we focus on a sparse block storage format where the dense blocks are placed on the main diagonal. This constraint leads to further optimisation possibilities, since we only need to encode the size of each block. For more details on these formats we refer the reader to canonical texts such as [12], [19].

While important, SpMV is hardly ever used in isolation. Typical algorithms require a combination of dense and sparse

basic linear algebra subroutines (BLAS). For example many numerical and scientific codes, such as the Conjugate Gradient Method [12], [14], follow this structure. An important property of iterative methods is that they do not update the input sparse matrix. This is a critical aspect in large scale problems (such as those typically solved in FEM), because explicit formation of updated matrix representation in memory, as well as the fill-in factor typically associated with direct methods can increase the storage and computational requirements by several orders of magnitude, depending on the problem instance.

Algorithm 1 Preconditioned Conjugate Gradient method

function CGM(M, P, b, y, tol, N_{max})
  $\vec{x} \leftarrow 0$, $\vec{r} \leftarrow b$
  $\vec{w} \leftarrow \mathbf{P}\vec{r}$ ▷ Applying preconditioner
  $\vec{s} \leftarrow \mathbf{M}\vec{w}$ ▷ Matrix-vector multiply
  $\varepsilon \leftarrow (\vec{r},\vec{r})$, $\mu \leftarrow (\vec{w},\vec{s})$, $\rho \leftarrow (\vec{w},\vec{r})$, $\alpha \leftarrow \rho/\mu$, $\beta \leftarrow 0$
  while $(N_{step} \leq N_{max})$ & $(\varepsilon < tol^2)$ **do**
    $\vec{p} \leftarrow \vec{w} + \beta\vec{p}$, $\vec{q} \leftarrow \vec{s} + \beta\vec{q}$ ▷ Vector arithmetic
    $\vec{x} \leftarrow \vec{x} + \alpha\vec{p}$, $\vec{r} \leftarrow \vec{r} - \alpha\vec{q}$
    $\vec{w} \leftarrow \mathbf{P}\vec{r}$ ▷ Applying preconditioner
    $\vec{s} \leftarrow \mathbf{M}\vec{w}$ ▷ Matrix-vector multiply
    $\varepsilon \leftarrow (\vec{r},\vec{r})$, $\mu \leftarrow (\vec{w},\vec{s})$, $\rho_{new} \leftarrow (\vec{w},\vec{r})$ ▷ Dot products
    $\beta \leftarrow \rho_{new}/\rho$, $\alpha \leftarrow \rho_{new}/(\mu - \rho_{new}\beta/\alpha)$
    $\rho \leftarrow \rho_{new}$
  end while
end function

Due to the associated challenges, SpMV has received much attention in the FPGA community. Earlier approaches focused on increasing the utilisation of on-chip resources, particularly for floating point SpMV. [15] proposes one of the first parametric designs for floating point SpMV and demonstrates how the flexibility of FPGAs can be used to achieve good performance compared to general purpose systems. More recently, the focus has shifted to efficient use of on-chip memory resources and DRAM bandwidth utilisation [5], [7], [9]. Recently, compression techniques have been proposed to improve the performance on memory bound matrices [8], [16]

The constant sparsity structure in the context of iterative methods has also been exploited to optimise FPGA architectures for SpMV [17]. Static one-off pre-processing techniques are cost-effective for FPGA implementations if they can lead either to a simplified architecture [5], [7], [18] or reduced communication overhead [8], [16]. Linear or log-linear pre-processing techniques with good performance in practice, such as the method used in this work for extracting matrix properties, have been found to be effective.

Recently, instance specific design methods have been proposed to explore FPGA specific optimisations more systematically: column based accelerators [4], instance specific methods for tuning architectural parameters [6] and compressing nonzero values [8] or metadata [16].

In this work we focus on higher-level methods which take into account the sparsity pattern of the given matrix. The technique of tuning SpMV kernels based on the sparsity pattern of the input matrix has also been explored on CPU systems with promising results [19], however it can have a much higher impact in reconfigurable FPGA based systems where we can adapt not only the algorithm and storage format but the architecture itself to maximise performance. This is exactly what we investigate in this work.

III. Architecture

We propose to exploit the flexibility of the FPGA to enable the acceleration of sparse matrix vector multiplication with block diagonal matrix structure in the context of iterative methods. The strategy is to use an optimised and customisable FPGA architecture in conjunction with corresponding resource and performance models (Section IV) and automated customisation techniques (Section V) to achieve high throughput, resource efficient designs for particular problem instances, starting from a characteristic input matrix, supplied by a domain expert.

The input matrix is a *variable size, dense block diagonal sparse matrix*: all nonzero elements can be grouped in dense matrix blocks, of potentially different orders, positioned along the main matrix diagonal, as shown in Figure 1. To support such matrices efficiently, the proposed architecture has three novel features:

1) *customisable trade-off between data and task level parallelism*, which enables efficient implementations for both large and small matrix blocks;
2) *independently customisable input and output types, compute types and mixed precision processing*, which enables careful balancing of I/O bandwidth, on-chip resources and computational accuracy;
3) *efficient partitioning and distribution strategy for matrix blocks*, which enables both the simplification of the proposed architecture and efficient, linear access pattern in off-chip memory.

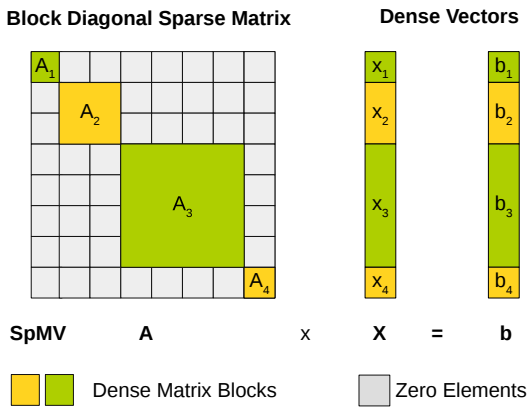

Fig. 1. Example block diagonal sparse matrix; matrix and vector blocks A_i, x_i, b_i are distributed to MPEs, as shown in Figure 2 and Figure 3

In this section we study the design space for the matrix-vector multiplication unit (MVMU) to identify resource efficient, high throughput implementations. We will illustrate these concepts on a FEM matrix, for which the size of each matrix block is induced by the geometric shape and polynomial order of its corresponding mesh element as shown

in Table I. We start from an efficient, parametric and customisable baseline implementation which facilitates design space exploration. By applying domain-specific customisation and instance specific design [6] we achieve a substantial improvement of resource efficiency compared to state of the art implementations [18]. In the case of our FEM case study, this increases the maximum supported problem size beyond what was previously possible on commercially available FPGAs.

The matrix-vector multiplication unit (MVMU) multiplies a block diagonal sparse matrix by a dense vector. This kernel is required on Line 10 of Algorithm 1. The MVMU architecture proposed in this section can be efficiently implemented on commercial FPGA architectures. The design is fully streaming and leads to a deep, fully pipelined architecture which maps well to the FPGA fabric. The design is resource efficient and makes effective use of C-slowing for the high latency double precision floating point accumulation operation. It also makes effective use of available on-chip resources and memory bandwidth by exploiting both data parallelism, through vectorisation, and task parallelism, through independent Matrix Processing Units or MPEs. Finally, the design is customisable: the number of MPEs (N_{MPE}), the vector width of each MPE (MPE_{width}), the depth of the accumulation buffer (MPE_D), and the input-output and compute types can be customised independently to support effective design space exploration. The benefits of this approach are illustrated in detail in the following section.

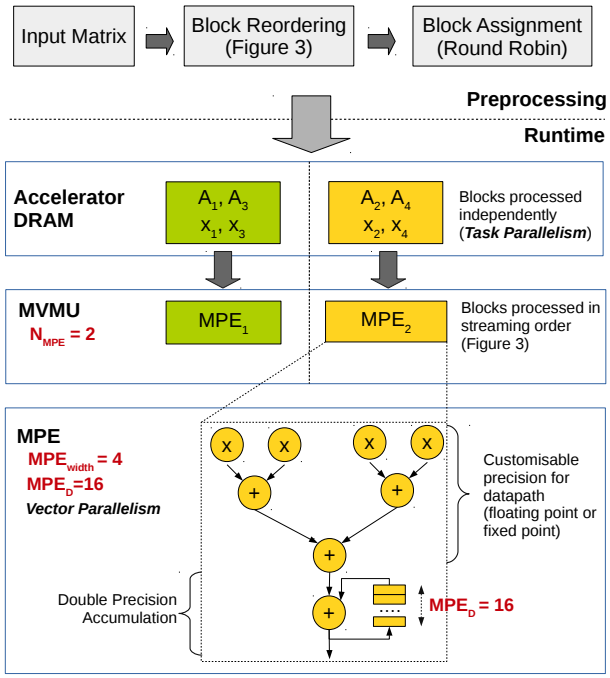

Fig. 2. Example matrix multiplier architecture with $N_{MPE} = 2, MPE_{width} = 4, MPE_D = 16$. Matrix blocks The blocks of the original matrix are reordered and assigned to each MPE on the CPU.

The architecture and operation of the MVMU are illustrated in Figure 2 and Figure 3. The operation is split among multiple MPEs as shown in Figure 2. Each MPE multiplies one block with the corresponding elements of the vector. The MPE processes the block in a column major order. Each MPE may process multiple columns of the same block concurrently. This increases the utilisation of the DRAM bandwidth, without requiring additional independent computation streams. We refer to the number of columns processed by a MPE as the *width* of the MPE (MPE_{width}). This is a compile time parameter in our design.

To enable the MPEs to process their assigned blocks in a column major order, each block is partitioned in vertical stripes as shown in Figure 3. The necessary reordering is performed on the CPU as described below. The width of a stripe is equal to MPE_{width}, which in the example of Figure 3 is 2. The final stripe may require zero padding, if the block size is not a multiple of MPE_{width}. This is a potential source of inefficiency in our design, and choosing an MPE_{width} which minimises the amount of padding is an important optimisation goal of the design space exploration process.

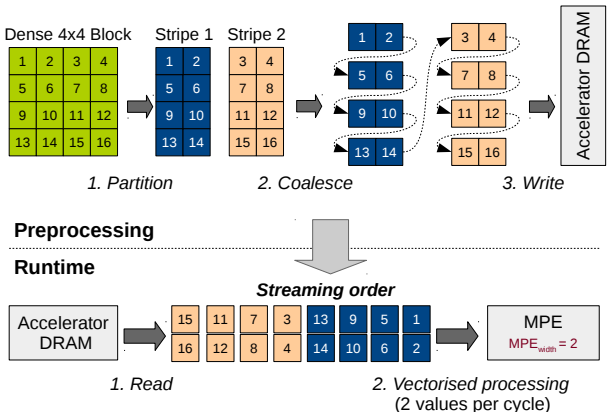

Fig. 3. Example distribution strategy for an architecture with $MPE_{width} = 2, N_{MPE} = 1$

Since matrix block sizes are different for each mesh element's geometric shape and polynomial order, the MVMU needs to be generic, efficiently supporting various matrix sizes. Therefore the partial sums of each row are accumulated in a variable depth FIFO. To function correctly this design requires that the maximum depth of the FIFO, MPE_D, is larger than the largest possible block. Additionally the effective depth of the FIFO must be configurable at runtime since it needs to exactly match the block size. This is achieved by updating the write and read address generators to wrap around based on the block size, instead of the statically configured MPE_D. This design functions correctly when the block size is smaller than MPE_D but greater than or equal to the floating point accumulator latency. The former is directly controllable by the method proposed in Section V, and the latter is the case for most problems of real practical interest. For example, for the Finite Element Method problem used as a case study in section VI, the block sizes already exceed 34 for all element types of practical interest.

To simplify the architecture, matrix data are pre-processed

on the CPU. To interface with CPU host code (such as open source implementations for the FEM method like Nektar++ [20]), we provide a method to add matrix blocks as they are being constructed. Our implementation creates a local copy of the block and prepares it for execution on the FPGA, as follows:

1) the block is padded, to ensure its size is a multiple of the MPE_{width};
2) the block is reordered so that the row-major access pattern is transformed to the striped access pattern required for MPE (as shown in Figure 3);
3) blocks are copied to on-board DRAM to utilise high bandwidth local interconnect;
4) after all blocks have been added, and a call to solve the resulting system is made, each block is distributed to the corresponding MPE.

We note that since all the operations above are performed locally to a matrix block (which is in practice small), they exhibit good locality of reference and their implementation is therefore efficient. The distribution heuristic is beyond the scope of the paper, so we leave this as an interesting future work opportunity. In the current implementation, we use a simple round robin based heuristic, where blocks are assigned in a circular order to each MPE. This heuristic is illustrated in Figure 2 where the first and third block are assigned to the first MPE and the second and fourth are assigned to the second MPE.

We implemented several optimisations to reduce the resource utilisation. Our design supports mixed precision computation: the type of the input vector and matrix to the design, the type used for vector operations and the types used for the reduction operation are all customisable. Since the implementation only operates at a reduced clock frequency (which is sufficient to utilise the available DRAM bandwidth), less aggressive pipelining is used which leads to substantial reduction in BRAM and Flip Flop resources.

IV. Resource Constrained Formulation

The flexibility of the baseline design enables domain and instance specific optimisations which leads to more efficient resource utilisation. In particular the following aspects of the baseline design are customisable and present interesting trade-off opportunities (parameters are also highlighted in Figure 2):

- N_{MPE} – increasing the number of MPEs allows us to saturate memory bandwidth while maintaining a reduced vector width, which could lead to increased efficiency; however since each MPE is connected to independent regions in memory, different data and command streams are required for each MPEs which leads to increased resource utilisation. This aspect is discussed in more detail in Section VI;
- MPE_{width} – the vector width of each MPE must be adjusted carefully to maximise the efficiency and DRAM bandwidth utilisation: too narrow vector units lead to poor DRAM bandwidth utilisation, too wide units may lead to low arithmetic efficiency depending on the block size. This aspect is discussed in more detail in Section VI;
- MPE_D – the depth of the accumulation buffer can be customised to support variable block sizes efficiently.

Based on these parameters, we propose a resource constrained analytical model for optimising the MVMU architecture for a particular matrix. The proposed model can be used to find optimised values for the architecture parameters based on the characteristics of a given matrix: the size of the blocks it contains BS_i and the number of blocks of each size NB_i.

First, as noted in Section III the depth of the accumulation buffer must be larger than or equal to the largest block size, for the C-slowed implementation to perform correctly:

$$\forall i \; MPE_D \geq BS_i \tag{1}$$

In addition, the memory bandwidth constraint must be taken into account: MPEs should not request more words per clock cycle than the width of the memory interface, B, for efficiency reasons:

$$N_{MPE} \times MPE_{width} \leq B \text{ (in words)} \tag{2}$$

We note that this equation ignores the transfer cost of the multiplicand and result vectors. The data size associated with each of these components is small: for every component of CG vectors, BS_i matrix components are streamed for each MPE, with BS_i reaching the value of several hundred for large polynomial orders.

The MVMU uses a streaming design so we assume the pipeline latency is negligible compared to the total number of cycles. The total number of cycles required to process a matrix in the MVMU is given by the number of cycles to process a block B_i, multiplied by the number of parallel block multiplication tasks available for that particular block size BS_i:

$$N_{Cycles} = \sum_i CyclesPerBlock(B_i) \times Tasks(B_i) \tag{3}$$

$$= \sum_i \left(\left\lceil \frac{BS_i}{MPE_{width}} \right\rceil \times BS_i \right) \times \left\lceil \frac{NB_i}{N_{MPE}} \right\rceil \tag{4}$$

This equation accounts for the fact that the block size BS_i may not be a multiple of the MPE_{width}, in which case additional cycles are required to process the stripe padding.

Finally, the resource usage model for the MVMU can be derived based on the cost of an adder R_+, a multiplier R_X, the accumulation buffer R_B, the width and depth of command and data FIFOs to and from DRAM for each MPE, R_{DRAM}, and the static resource utilisation for components such as PCIe and DDR3 interface and controllers R_{static}:

$$R = R_{Static} + N_{MPE} \times (MPE_{width} \times (R_+ + R_X) + R_B + R_{DRAM}) \tag{5}$$

As we discuss in Section VI, R_{DRAM} itself is influenced by MPE_{width} as the additional buffering introduced between the DRAM interface and the kernel interface is directly

proportional to the lowest common multiple (LCM) of the two:

$$R_{DRAM} \propto FIFO_{width} \times FIFO_{depth} \times \text{LCM}(\text{Burst Size}, MPE_{width})$$

Therefore, the problem of optimising the MVMU for a particular mesh configuration reduces to the problem of finding the values of N_{MPE}, MPE_{width} and MPE_D to minimise N_{cycles} from Eq. 5, subject to the functional constraints given by Eq. 1 and Eq. 2 and the resource utilisation constraint $R_{MVMU} \leq S_{Resources}$ where $S_{Resources}$ represents the spare resources available for the MVMU.

Applying this model and all proposed optimisations substantially reduces the resource usage for the MVMU, as shown in Section VI, enabling us to fit larger meshes than previously possible [18].

V. Automated Customisation Method

The parametric architecture presented in Section III together with the resource constrained optimisation method in Section IV can be used to automatically generate or select at runtime the most efficient architecture for the MVMU based on the properties of the input matrix: the block sizes, BS_i, and number of blocks of each size, NB_i. For a block diagonal sparse matrix, these parameters can be determined using Algorithm 2. We assume a function $rowSpan$ which can return the column index of the first and the last nonzeros in a row. The implementation of $rowSpan$ depends on the storage format of A. Any sparse storage format can be used which supports the $sortEntries$ and $rowSpan$ functions efficiently, such as CSR, CSC etc. Algorithm 2 assumes that the first and last element in a block are not zero, which is guaranteed by the block diagonal structure of the matrix.

Algorithm 2 Extract BS_i, NB_i for the sparse matrix A

```
1:  function ExtractBlockMatrixParams(A)
2:      SortEntries(A)                    ▷ Sort by row and column index
3:      occ ← {}                          ▷ Dictionary of BS_i and NB_i
4:      for i ∈ 0 ... A.nrows do
5:          (first, last) ← RowSpan(i)
6:          bs ← last − first             ▷ Assume new block size
7:          startPosition ← i             ▷ Upper left corner of block
8:          while i − startPosition < bs do
9:              if last > startPosition + bs then
10:                 ERROR! Not a block diagonal matrix
11:             else if first < startPosition then
12:                 ERROR! Not a block diagonal matrix
13:             end if
14:             i ← i + 1
15:             (first, last) ← RowSpan(i)
16:         end while
17:         occ[bs] ← occ[bs] + 1         ▷ update block count
18:     end for
19:     return occ                        ▷ return the frequency map of block sizes
20: end function
```

First, the algorithm sorts the nonzeros of A in increasing order of their row and column index. Sorting is relatively inexpensive as a one-time pre-processing step, with time complexity $O(N_{nnz} \times log(N_{nnz}))$ where N_{nnz} is the number of nonzero elements of A, but even so, it is very common in practice for sparse matrices to contain entries already ordered in this fashion, since it is a pre-requisite of efficient access and manipulation in many algorithms. In this case the sorting step can be skipped entirely.

Second, the algorithm performs a linear sweep for blocks. The outer loop finds the dimension of the block bs using the $RowSpan$ function while the inner loop traverses the next $bs - 1$ rows to ensure that no element resides outside the inferred block size. If an offending element is found residing outside the inferred block (either before or after the block), the algorithm quits and assumes the matrix is not block-diagonal. In this approach explicit zero entries may be stored in the inferred blocks but there is nothing which can be done in this regards, if we assume a block-diagonal matrix.

The flow proposed in this work combines the parametric architecture presented in Section III, the resource constrained optimisation method shown in Section IV and Algorithm 2 to enable the customisation and runtime selection of problem specific MVMU architectures. The resulting flow is shown in Figure 4 and comprises two main stages *tuning* and *runtime selection*.

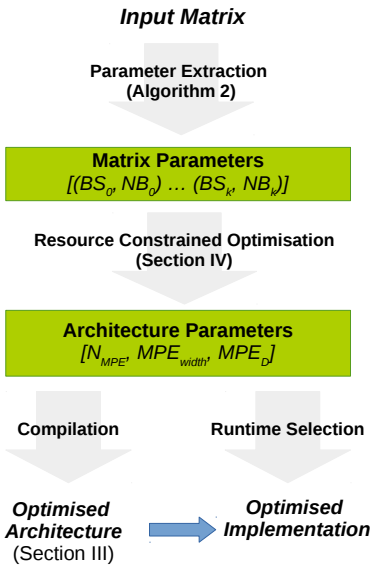

Fig. 4. Overview of the proposed approach

During the *tuning* stage, the optimal set of architectures is determined from a set of input block diagonal sparse matrices, which are considered representative for the problem or application domain under study. For every sparse matrix in the benchmark: *1)* the matrix parameters BS_i and NB_i are extracted, *2)* based on these parameters resource constrained optimisation is applied to determine the optimal architecture configuration, *3)* the optimal architecture is synthesised. Finally, the generated architectures and a driver program are packaged as a shared library, available for loading during the runtime stage.

During the *runtime* stage, the appropriate architecture must be configured on the FPGA. This requires, again, to determine the matrix parameters BS_i and NB_i the corresponding optimal values of N_{MPE}, MPE_{width}, MPE_D, but in this case based on the actual runtime values of the sparse matrix. As explained, this process is inexpensive, and the cost is amortised over the long running time for iterative applications such as the FEM. Finally, the selected architecture can be loaded on the FPGA using runtime reconfiguration.

VI. Evaluation

We evaluate our approach in the context of the Finite Element Method, a classical HPC problem. We observe significant variance in throughput, compute efficiency and resource utilisation based on matrix block size and architectural parameters. This insight can be used to optimise the MVMU accordingly, providing more spare resources for other critical components: our method automatically detects the optimal values for the architecture parameters for a particular sparse block diagonal input matrix. In the case of the Finite Element Method, this leads to an ability to support larger problem sizes on a single device and therefore reduced inter-device communication volume, which translates to increased performance.

The Finite Element Method [1] is used to solve Partial Differential Equations (PDE) on large scale geometric domains, which are used in many fields of engineering. The Finite Element Method operates on meshes which represent the geometrical domains by splitting them into *elements*, simple geometrical shapes (e.g. tetrahedra) with an unstructured adjacency graph. The solution to the PDE is approximated with a piece-wise polynomial defined at every element separately.

The spectral/hp FEM method used in our evaluation discretises the PDE on each element into a dense matrix, whose size depends on the geometrical shape of the element and the polynomial order used for approximating the PDE solution. Due to this discretisation strategy, the whole mesh can be represented as a block-diagonal matrix with matrix blocks of variable size. Exact formulas for the block sizes depend on element shape and polynomial order P, with the typical dimension of a sparse matrix block ranging between a few tens and a few hundreds of elements, as shown in Table I.

TABLE I
Block sizes for different polynomial orders and geometrical shapes present in typical FEM meshes

Element type	P=1	P=2	P=3	P=4	P=5	P=6
Tetrahedron	4	10	20	34	52	74
Prism	6	18	38	66	102	146
Hexahedron	8	26	56	98	152	218

We use the *local matrix approach* [1] which represents the problem with the block-diagonal matrix described above at every evaluation of a matrix-vector multiply. For higher polynomial orders, where acceleration is most needed, the local matrix approach becomes more computationally efficient than other approaches, since it exploits the more regular structure of the block-diagonal dense matrix storage thus leading to more efficient memory access and use of parallelism [21].

We implement the proposed architecture using Maxeler MaxCompiler 2015.2 and the MaxJ dataflow language. All resource utilisation results correspond to post place and route resource usage annotations provided by the Maxeler MaxCompiler. We provide a CPU interface in C++11 which allows the addition and pre-processing of matrix blocks as described in Section III.

We use a Maxeler MPCX Dataflow node. The system properties are summarised in Table II. It consists of a CPU node and a DFE node. The two are connected via Infiniband through a Mellanox FDR Infiniband switch.

TABLE II
System Properties

CPU	Dual Intel Xeon E5-2640
CPU Cache	15 MB
CPU DRAM Bandwidth	42.6 GB/s (Peak)
CPU DRAM Capacity	64GB DDR3-1333
FPGA	Stratix V 5SGSMD8N1F45C2
FPGA DRAM Bandwidth	58 GB/s (Achieved)
FPGA DRAM Capacity	48 GB
CPU to FPGA Bandwidth	2 GB/s

We illustrate the benefits of the optimisations and design method proposed in Section III by providing a study of customisation opportunities on the NACA 1L FEM mesh [22] which has been used as a benchmark in previous work [18], [23]. The NACA 1L mesh is an unstructured mesh with 58728 tetrahedral elements of block size 20x20 and 11549 prismatic elements of block size 38x38 (at polynomial order $P = 3$). In practice, larger values of P are used for large scale case studies (typically 4–7), which as explained would make our approach even more resource efficient, due to the large corresponding block sizes. However for this section we use $P = 3$ to provide a comparison with prior work [18], [23]. We study both the computational efficiency and the resource efficiency of the proposed design.

From the computational efficiency perspective, we notice a complex interaction between DRAM bandwidth utilisation, resource utilisation and compute efficiency induced by problem specific properties such as block order and required accuracy. For example higher vector widths may lead to better bandwidth utilisation but reduced compute efficiency for small blocks.

More generally, depending on the polynomial order, appropriate vector widths can be used to reduce on-chip resource usage while maintaining throughput, thus increasing overall resource efficiency. As shown in Figure 5, for the NACA 1L mesh, the utilisation of vector units varies considerably with MPE_{width}. The compute efficiency can be defined as

$$E = \frac{\text{Useful Operations}}{\text{Total Operations}} = \frac{\sum_i BS_i^2 \times NB_i}{\sum_i \left\lceil \frac{BS_i}{MPE_{width}} \right\rceil \times MPE_{width} \times BS_i \times NB_i}$$

TABLE III

POST PLACE AND ROUTE RESULTS FOR THE MVMU ON A STRATIX V D8 CHIP USING MAXELER MAXCOMPILER 2015.2. RESOURCES USED ARE SHOWN BOTH AS A NUMBER AND A PERCENT OF THE TOTAL RESOUCES AVAILABLE (%). DATA TYPES FOR MATRIX INPUT (M_{in}), VECTOR INPUT (V_{in}) AND COMPUTE (C) ARE SHOWN AS DOUBLE PRECISION (D), SINGLE PRECISION (S) OR INTEGER / FRACTION BITS FOR FIXED POINTS, OR EXPONENT / MANTISSA BITS FOR FLOATING POINT

Id	N_{MPE}	MPE_{width}	M_{in}	V_{in}	C	**Fixed**	**Logic** ($U/\%$)	**DSP** ($U/\%$)	**BRAM** ($U/\%$)
1	3	24	S	D	S	F	97 / 4	72 / 3.67	1058 / 41.22
2	3	24	S	S	S	F	92 / 4	72 / 3.67	1001 / 38.99
3	3	24	S	D	D	F	109 / 4	288 / 14.67	1061 / 41.33
4	3	24	S	D	11 32	F	104 / 4	144 / 7.34	1061 / 41.33
5	3	24	6 58	D	6 58	T	86 / 3	576 / 29.34	1094 / 42.62
6	1	8	S	D	S	F	67 / 3	8 / 0.41	551 / 21.46
7	1	10	S	D	S	F	83 / 3	10 / 0.51	1323 / 51.54
8	1	20	S	D	S	F	85 / 3	20 / 1.02	1347 / 52.47
9	1	30	S	D	S	F	87 / 3	30 / 1.53	1371 / 53.41
10	1	48	S	D	S	F	78 / 3	48 / 2.45	570 / 22.20
11	1	96	S	D	S	F	90 / 3	96 / 4.89	686 / 26.72

and it accounts for the fact that vector units within an MPE may not be processing useful values at all times, but also zero padding entries. This fact is unavoidable because the block size varies with the problem specification and even within the same matrix, blocks of various sizes are used. However the width of the vector unit remains fixed, in the absence of partial or run-time reconfiguration.

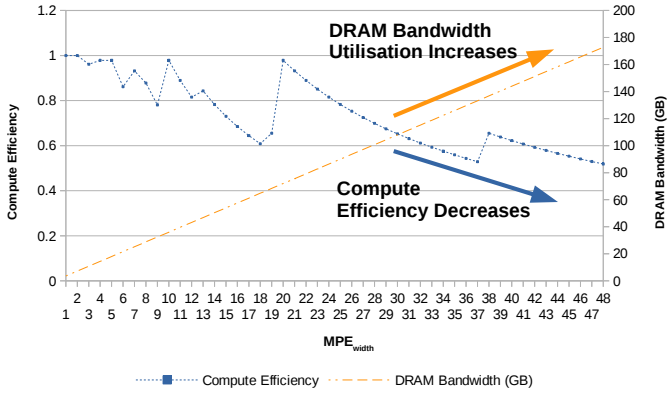

Fig. 5. Efficiency and DRAM bandwidth versus MPE_{width} for the NACA 1L mesh with P = 3 and block sizes of 20x20, 38x38

Furthermore MPE_{width} is also constrained from a resource utilisation perspective by the strategy used for buffering data and commands sent to and from DRAM. In the case of the Maxeler MaxCompiler, the depth of FIFOs used is the largest common multiple of the width of the input request and the DRAM burst size, by default 384 bytes. This makes certain input widths inefficient to implement from a resource utilisation perspective, as shown in Table III: Architecture 10 with $MPE_{width} = 48$ consumes less than half the BRAM resources of Architecture 7 with $MPE_{width} = 10$. Clearly there is a substantial resource saving benefit to the latter option, but the computational efficiency on the NACA 1L mesh ($P = 3$, block sizes of 20x20, 38x38) is very poor as illustrated in Figure 5: 0.5 for an MPE_{width} of 48 compared to more than 0.9 for an MPE_{width} of 10.

In addition, Figure 5 illustrates the tension between fully utilising off-chip memory bandwidth and achieving a good utilisation for on-chip resources: increasing the vector width directly increases the utilisation of off-chip bandwidth, but may lead to reduced efficiency due to extra cycles spent on processing zeros, when the block size is smaller than the vector width or the block size is not a multiple of MPE_{width}.

From the resource utilisation perspective, it is important to emphasise that this is not simply a case of aggressively configuring the MVMU unit to deal with largest possible block sizes: as shown in Figure 5 this approach may lead to inefficient resource utilisation, leaving less on-chip resources for other application kernels which may eventually result in reduced overall performance.

Figure 6 shows the throughput per BRAM of two sample architectures from Table III: Architecture 1 with $N_{MPE} = 3, MPE_{width} = 24$ and Architecture 11 with $N_{MPE} = 1, MPE_{width} = 96$. This captures the scalability of the design with respect to the bounding resource, BRAMs. It illustrates the effective trade-off between vector style and task style parallelism within the MVMU. For smaller block sizes, the task style parallelism achieves higher efficiency and as a result, even though the resource utilisation is considerably higher (1058 BRAMs for Architecture 1 vs 686 BRAMs for Architecture 11), the task parallel approach achieves a better utilisation of on-chip resources. For the larger block sizes, where there is sufficient data parallelism within a row, vector style parallelism is more efficient from a resource utilisation point of view since it requires significantly fewer resources for the memory controller. This fact can also be corroborated with Figure 5.

An additional possibility for reducing resource utilisation is the use of wordlength optimisation. For the purpose of maximising resource efficiency our implementation supports arbitrary width fixed and floating point inputs as well as mixed precision computation, with the more critical parts of the computation, performed in higher precision. The reference CPU implementation of the Nektar++ framework and our baseline design uses double precision floating point arithmetic for all operations. Compared to this baseline design, the 43 bit double precision implementation, Architecture 4 of Table III, achieves a substantial reduction in DSP usage: half the number

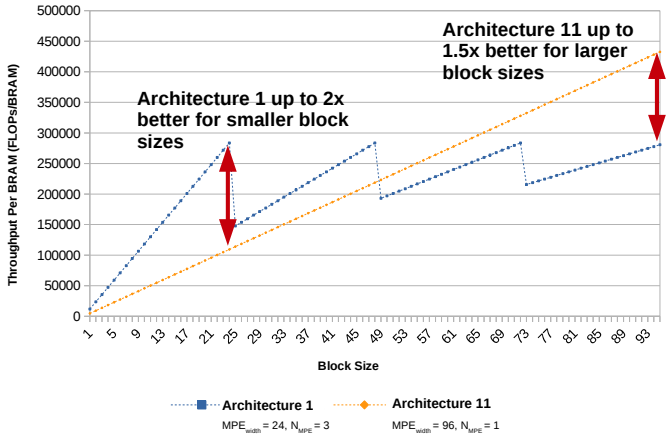

Fig. 6. Throughput per BRAM for two architectures shown in Table III

of DSPs are used compared to the full double precision version. While DSP utilisation is not a bounding resource in the current design, in other compute intensive applications, this saving can prove important. However we note that fixed point implementation does not represent an entirely viable option for FEM problems due to the large wordlength required for convergence, 64 bit. Our numerical experiments with Nektar++ shows that 64 bits fixed are required for the iterative solver to converge. The large wordlength leads to a significantly increased DSP utilisation, and a slight increase in BRAM usage but a slight reduction in logic usage.

Table IV shows that for the NACA 1L mesh for the MVMU unit a substantial saving can be achieved in resource usage by deploying the architecture proposed in this work: the MVMU unit uses almost 10 times less BRAMs and about half the logic resources compared to a similar, state of the art design implemented in [18]. In fact the unit itself is so optimised that the overhead for the periphery units, such as PCIe and Memory Controller, dominate resource utilisation. As Table IV shows the memory controller occupies more than 30 times the area (in BRAMs) of the MVMU. However, we do not believe this is a scalability concern since vendors can provide Hard Memory Controllers to reduce resource utilisation and to improve throughput for memory interfaces to be on par with modern GPU architectures.

TABLE IV
RESOURCE UTILISATION IMPROVEMENTS COMPARED TO PRIOR WORK

Kernel	BRAMs	LUTs	FFs	DSPs
MVMU, 3 MPEs [18]	201	105129	145147	288
MVMU, 3 MPEs (this work)	21	54340	61923	288
Memory Controller	652	17165	64581	
PCIe	100	6713	7828	
FIFOs	184	568	709	

Finally, Figure 7 shows the expected impact of applying the proposed method to an FPGA implementation of the FEM framework Nektar++ [18], [20]. In addition to the matrix vector multiplier, in [18], the vector scatter/gather units, preconditioner and linear algebra units require BRAM resources. As a result, the original design overmaps significantly on BRAMs for the NACA One Layer mesh. The architecture proposed in this work and implemented with the same parameters as in [18] is already close to fitting on chip, as it overmaps by only 45 BRAMs. However by carefully applying the trade-off between vector and task level parallelism we can identify configurations (such as $N_{MPE} = 2$, $MPE_{width} = 48$) which fit on chip, and achieve speedup of up to 3 times over an optimised multi-threaded implementation from the Nektar++ framework [20].

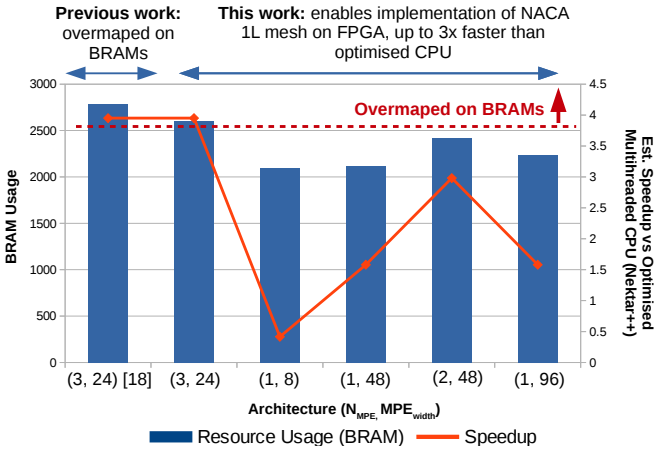

Fig. 7. Projected results for applying the proposed approach to accelerate the FEM framework Nektar++

VII. CONCLUSION

In this work we propose an architecture and an automated customisation method to detect and optimise the architecture for sparse matrices with block diagonal structure, which arise often in practice. We evaluate the proposed approach in the context of the spectral/hp Finite Element Method, a classic High Performance Computing problem. The efficiency of the proposed architecture combined with the effectiveness of the proposed customisation method reduces BRAM resource utilisation by as much as 10 times, while achieving identical throughput with existing state of the art designs and requiring minimal development effort from the end user.

Opportunities for future work include developing a variety of generic pattern detection methods, including additional benchmarks for evaluation, and exploring further optimisations, such as compression techniques that improve memory bandwidth utilisation [8].

ACKNOWLEDGMENTS

This work is supported in part by the EPSRC under grant agreements EP/L016796/1, EP/L00058X/1 and EP/I012036/1, by the Maxeler University Programme, by the HiPEAC NoE, by Altera, and by Xilinx.

REFERENCES

[1] G. Karniadakis and S. Sherwin, *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press, 2013.

[2] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix–vector Multiplication on Emerging Multicore Platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.

[3] Y. Zhang, Y. H. Shalabi, R. Jain, K. K. Nagar, and J. D. Bakos, "FPGA vs. GPU for sparse matrix vector multiply," in *Proc. ICFPT*, 2009.

[4] R. Dorrance, F. Ren, and D. Marković, "A Scalable Sparse Matrix-Vector Multiplication Kernel for Energy-efficient Sparse-BLAS on FPGAs," in *Proc. FPGA*, 2014, pp. 161–170.

[5] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication," in *Proc. FPGA*, 2014.

[6] P. Grigoras, P. Burovskiy, and W. Luk, "CASK: Open-source custom architectures for sparse kernels," in *Proc. FPGA*, 2016, pp. 179–184.

[7] G. Chow, P. Grigoras, P. Burovskiy, and W. Luk, "An Efficient Sparse Conjugate Gradient Solver Using a Benes Permutation Network," in *Proc. FPL*, 2014.

[8] P. Grigoras, P. Burovskiy, E. Hung, and W. Luk, "Accelerating SpMV on FPGAs by Compressing Nonzero Values," in *Proc. FCCM*, 2015.

[9] K. Townsend and J. Zambreno, "Reduce, Reuse, Recycle (R 3): A design methodology for Sparse Matrix Vector Multiplication on reconfigurable platforms," in *Proc. ASAP*, 2013.

[10] K. K. Nagar and J. D. Bakos, "A Sparse Matrix Personality for the Convey HC-1," in *Proc. FCCM*, 2011.

[11] L. Zhuo and V. K. Prasanna, "Sparse Matrix-Vector Multiplication on FPGAs," in *Proc. FPGA*, 2005.

[12] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Society for Applied and Industrial Mathematics, 2003.

[13] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *High Performance Computing and Communications*. Springer, 2005, pp. 807–816.

[14] R. Fletcher, "Conjugate gradient methods for indefinite systems," in *Numerical Analysis*. Springer, 1976, pp. 73–89.

[15] L. Zhuo, G. R. Morris, and V. K. Prasanna, "Designing Scalable FPGA-based Reduction Circuits Using Pipelined Floating-point Cores," in *Proc. ISPDP*, 2005.

[16] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a Universal FPGA Matrix-Vector Multiplication Architecture," in *Proc. FCCM*, 2012.

[17] M. DeLorimier and A. DeHon, "Floating-point Sparse Matrix-Vector Multiply for FPGAs," in *Proc. FPGA*, 2005.

[18] P. Burovskiy, P. Grigoras, S. J. Sherwin, and W. Luk, "Efficient Assembly for High Order Unstructured FEM Meshes," in *Proc. FPL*, 2015.

[19] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521.

[20] C. D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot *et al.*, "Nektar++: An open-source spectral/hp element framework," *Computer Physics Communications*, vol. 192, pp. 205–219, 2015.

[21] P. E. J. Vos, S. J. Sherwin, and R. M. Kirby, "From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations," *J. Comput. Phys.*, vol. 229, no. 13, pp. 5161–5181, Jul. 2010.

[22] J. S. Chow, G. G. Zilliac, and P. Bradshaw, "Mean and turbulence measurements in the near field of a wingtip vortex," *AIAA journal*, vol. 35, no. 10, pp. 1561–1567, 1997.

[23] J. Hu, S. F. Quigley, and A. Chan, "An element-by-element preconditioned conjugate gradient solver of 3d tetrahedral finite elements on an FPGA coprocessor," in *Proc. FPL*. IEEE, 2008, pp. 575–578.